

Engenharia de Software II

A photograph showing a person's legs and feet as they sweep a light-colored wooden floor with a teal-handled broom. A pile of dark dust is visible on the floor where the broom has been sweeping. In the background, a white rug is partially visible.

Clean Code

Aula 02

Referência

Robert C. Martin Series

Clean Code

A Handbook of Agile Software Craftsmanship

Robert C. Martin



Foreword by James O. Coplien

E esse código? O que faz?

```
● ● ●  
1  async completeUserData(  
2    id: string,  
3    data: UpdateUserInput,  
4    photo?: FileUpload,  
5  ): Promise<User> {  
6    const foundUser: User = await this.userRepository.findOne(id);  
7  
8    if (!foundUser) {  
9      throw new NotFoundException(consts.USER_NOT_FOUND);  
10   }  
11  
12   if (foundUser.password) {  
13     throw new UnauthorizedException(consts.USER_PASS_ALREADY_DEFINED);  
14   }  
15  
16   if (!foundUser.status || !(foundUser.status === Status.ATIVO)) {  
17     throw new UnauthorizedException(consts.USER_NOT_ACTIVATED);  
18   }  
19  
20   if (!foundUser.roles) {  
21     foundUser.roles = [await this.rolesService.findByName(RolesEnum.USER)];  
22   }  
23  
24   if (!data.password) {  
25     throw new UnauthorizedException(consts.USER_PASS_EMPTY);  
26   }  
27  
28   if (data.email) {  
29     const foundAnotherUserWithThisEmail: User = await this.userRepository.findOne(  
30       {  
31         id: Not(id),  
32         email: data.email,  
33       },  
34     );  
35     if (foundAnotherUserWithThisEmail) {  
36       throw new UnauthorizedException(consts.USER_EMAIL_ALREADY_EXISTS);  
37     }  
38   }  
39 }
```

```
39  
40   if (data.phone) {  
41     const foundAnotherUserWithThisPhone: User = await this.userRepository.findOne(  
42       {  
43         id: Not(id),  
44         phone: data.phone,  
45       },  
46     );  
47     if (foundAnotherUserWithThisPhone) {  
48       throw new UnauthorizedException(consts.USER_PHONE_ALREADY_EXISTS);  
49     }  
50   }  
51  
52   if (photo) {  
53     const docFile: File = await this.fileService.uploadFile({  
54       uploadFile: photo,  
55       isPrivate: false,  
56       isImage: true,  
57     });  
58     foundUser.photo = docFile;  
59   }  
60  
61   return this.userRepository.save({  
62     ...foundUser,  
63     ...data,  
64     roles: foundUser.roles,  
65   });  
66 }
```

Solução de um Segundo Dev

```
1  async completeUserData(  
2    id: string,  
3    data: UpdateUserInput,  
4    photo?: FileUpload,  
5  ): Promise<User> {  
6    const foundUser: User = await this.userRepository.findOne(id);  
7  
8    // Valida se o usuário a ser atualizado existe  
9    if (!foundUser) {  
10      throw new NotFoundException(consts.USER_NOT_FOUND);  
11    }  
12  
13   // Valida se o usuário já possui senha  
14   if (foundUser.password) {  
15     throw new UnauthorizedException(consts.USER_PASS_ALREADY_DEFINED);  
16   }  
17  
18   // Valida se o usuário está ativo  
19   if (!foundUser.status || !(foundUser.status === Status.ATIVO)) {  
20     throw new UnauthorizedException(consts.USER_NOT_ACTIVATED);  
21   }  
22  
23   // Se o usuário não tiver roles, adiciona a role de usuário padrão  
24   if (!foundUser.roles) {  
25     foundUser.roles = [await this.rolesService.findByName(RolesEnum.USER)];  
26   }  
27  
28   // Valida se está sendo atualizada a senha  
29   if (!data.password) {  
30     throw new UnauthorizedException(consts.USER_PASS_EMPTY);  
31   }  
32  
33   // Valida se o email a ser atualizado já existe  
34   if (data.email) {  
35     const foundAnotherUserWithThisEmail: User = await this.userRepository.findOne(  
36       {  
37         id: Not(id),  
38         email: data.email,  
39       },  
40     );  
41     if (foundAnotherUserWithThisEmail) {  
42       throw new UnauthorizedException(consts.USER_EMAIL_ALREADY_EXISTS);  
43     }  
44   }  
45  
46   // Valida se o phone a ser utilizado já existe  
47   if (data.phone) {  
48     const foundAnotherUserWithThisPhone: User = await this.userRepository.findOne(  
49       {  
50         id: Not(id),  
51         phone: data.phone,  
52       },  
53     );  
54     if (foundAnotherUserWithThisPhone) {  
55       throw new UnauthorizedException(consts.USER_PHONE_ALREADY_EXISTS);  
56     }  
57   }  
58  
59   // Atualiza a foto se for enviada  
60   if (photo) {  
61     const docFile: File = await this.fileService.uploadFile({  
62       uploadFile: photo,  
63       isPrivate: false,  
64       isImage: true,  
65     });  
66     foundUser.photo = docFile;  
67   }  
68  
69   return this.userRepository.save({  
70     ...foundUser,  
71     ...data,  
72     roles: foundUser.roles,  
73   });  
74 }
```

```
41     if (foundAnotherUserWithThisEmail) {  
42       throw new UnauthorizedException(consts.USER_EMAIL_ALREADY_EXISTS);  
43     }  
44   }  
45  
46   // Valida se o phone a ser utilizado já existe  
47   if (data.phone) {  
48     const foundAnotherUserWithThisPhone: User = await this.userRepository.findOne(  
49       {  
50         id: Not(id),  
51         phone: data.phone,  
52       },  
53     );  
54     if (foundAnotherUserWithThisPhone) {  
55       throw new UnauthorizedException(consts.USER_PHONE_ALREADY_EXISTS);  
56     }  
57   }  
58  
59   // Atualiza a foto se for enviada  
60   if (photo) {  
61     const docFile: File = await this.fileService.uploadFile({  
62       uploadFile: photo,  
63       isPrivate: false,  
64       isImage: true,  
65     });  
66     foundUser.photo = docFile;  
67   }  
68  
69   return this.userRepository.save({  
70     ...foundUser,  
71     ...data,  
72     roles: foundUser.roles,  
73   });  
74 }
```

Trocando os condicionais por métodos

```
1  async completeUserData(
2    id: string,
3    data: UpdateUserInput,
4    photo?: FileUpload,
5  ): Promise<User> {
6    const foundUser: User = await this.userRepository.findOne(id);
7
8    this.validateIfUserExist(foundUser);
9    this.validateIfUserHasPassword(foundUser);
10   this.validateIfIsActive(foundUser);
11   this.validateOrAddDefaultRole(foundUser);
12
13  this.validateIfIsUpdatingPassword(data.password);
14  this.validateIfIsEmailAvailable(data.password);
15  this.validateIfIsPhoneAvailable(data.phone);
16
17  this.updatePhotoIfExist(photo);
18
19  return this.userRepository.save({
20    ... foundUser,
21    ... data,
22    roles: foundUser.roles,
23  });
24 }
```

Agrupando os métodos por escopo

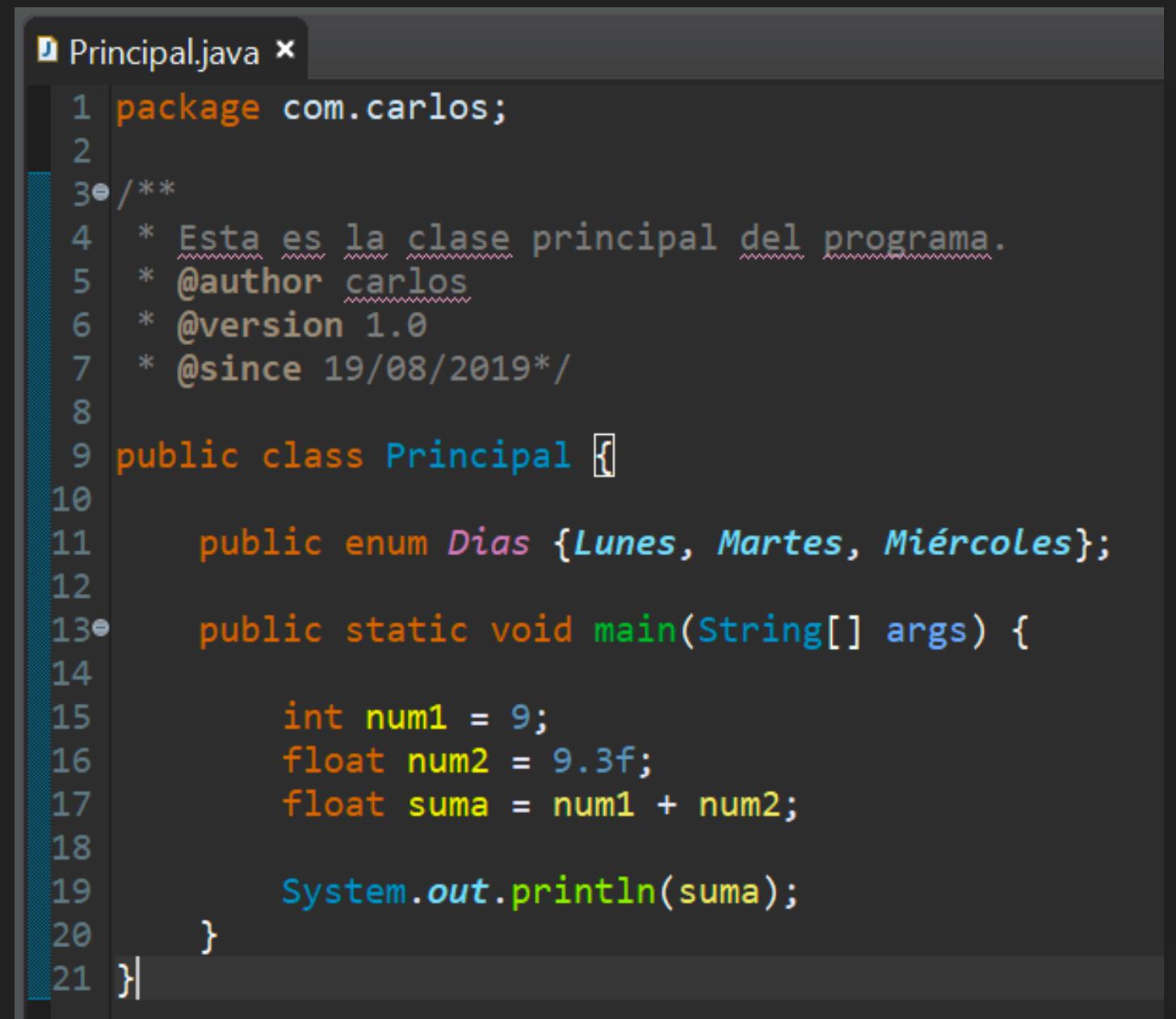
```
● ● ●  
1  id: string,  
2  data: UpdateUserInput,  
3  photo?: FileUpload,  
4 ): Promise<User> {  
5  const foundUser: User = await this.userRepository.findOne(id);  
6  
7  this.validateUserOrFail(foundUser);  
8  
9  this.validateInputDataOrFail(data);  
10  
11 this.updatePhotoIfExist(photo);  
12  
13 return this.userRepository.save({  
14   ... foundUser,  
15   ... data,  
16   roles: foundUser.roles,  
17 });  
18 }
```

Comentários

Comentários - Adicionar ou evitar

Comentários com nomes e informações dos autores do código?

✓ Adicionar

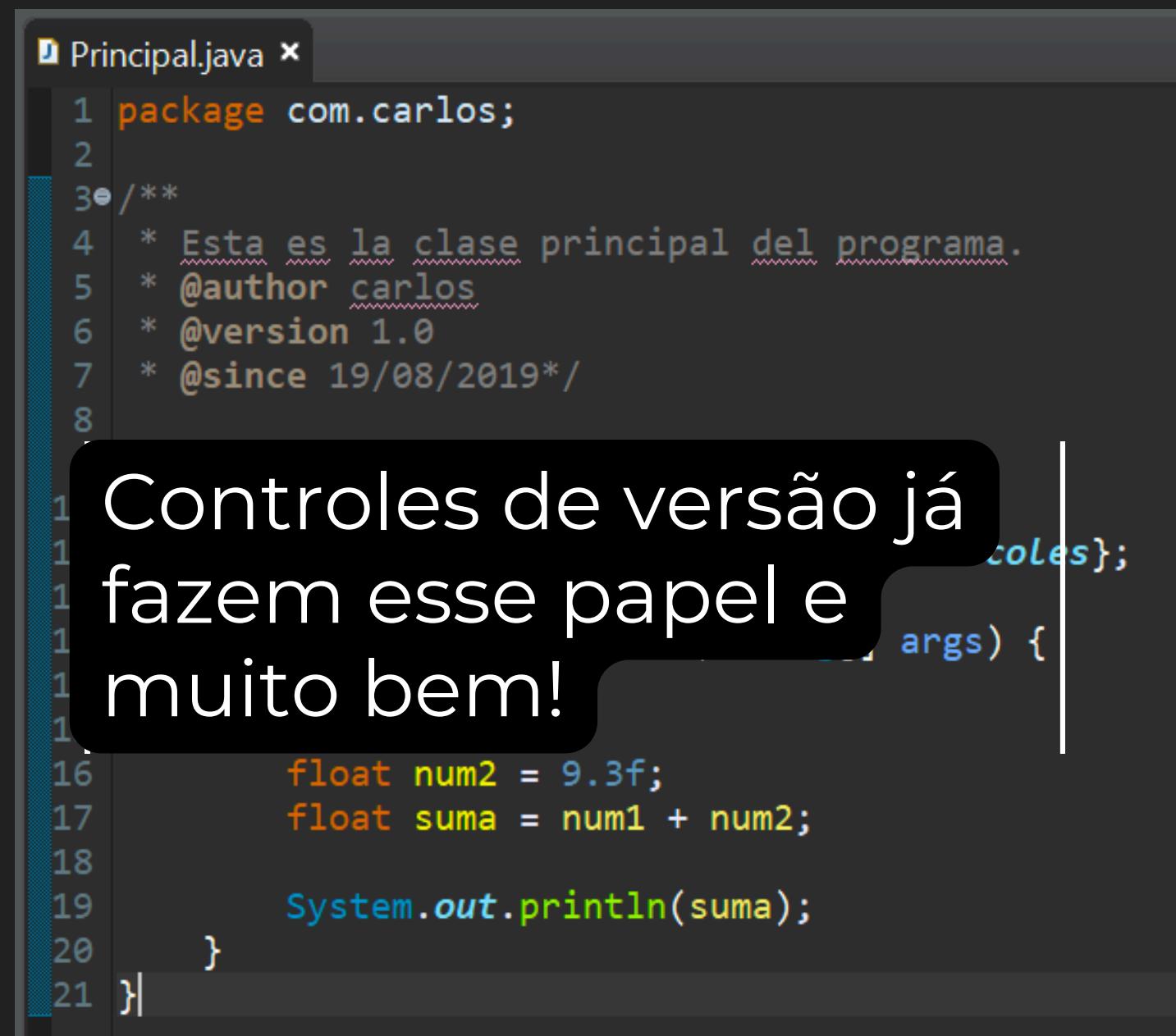


```
1 package com.carlos;
2
3 /**
4  * Esta es la clase principal del programa.
5  * @author carlos
6  * @version 1.0
7  * @since 19/08/2019
8
9 public class Principal {
10
11     public enum Dias {Lunes, Martes, Miércoles};
12
13     public static void main(String[] args) {
14
15         int num1 = 9;
16         float num2 = 9.3f;
17         float suma = num1 + num2;
18
19         System.out.println(suma);
20     }
21 }
```

✗ Evitar

Comentários - Adicionar ou evitar

Comentários com nomes e informações dos autores do código?  Adicionar



```
1 package com.carlos;
2
3 /**
4  * Esta es la clase principal del programa.
5  * @author carlos
6  * @version 1.0
7  * @since 19/08/2019*/
8
9
10 public class Principal {
11     public static void main(String[] args) {
12         float num1 = 5.5f;
13         float num2 = 9.3f;
14         float suma = num1 + num2;
15         System.out.println(suma);
16     }
17 }
18
19
20
21 }
```

Controles de versão já fazem esse papel e muito bem!



Evitar

- Informações de autor
-

Comentários - Adicionar ou evitar

Comentar os métodos como:



```
1  /**
2   *
3   * @returns o nome do usuário
4   */
5  public getName(): string {
6    return this._name;
7 }
```



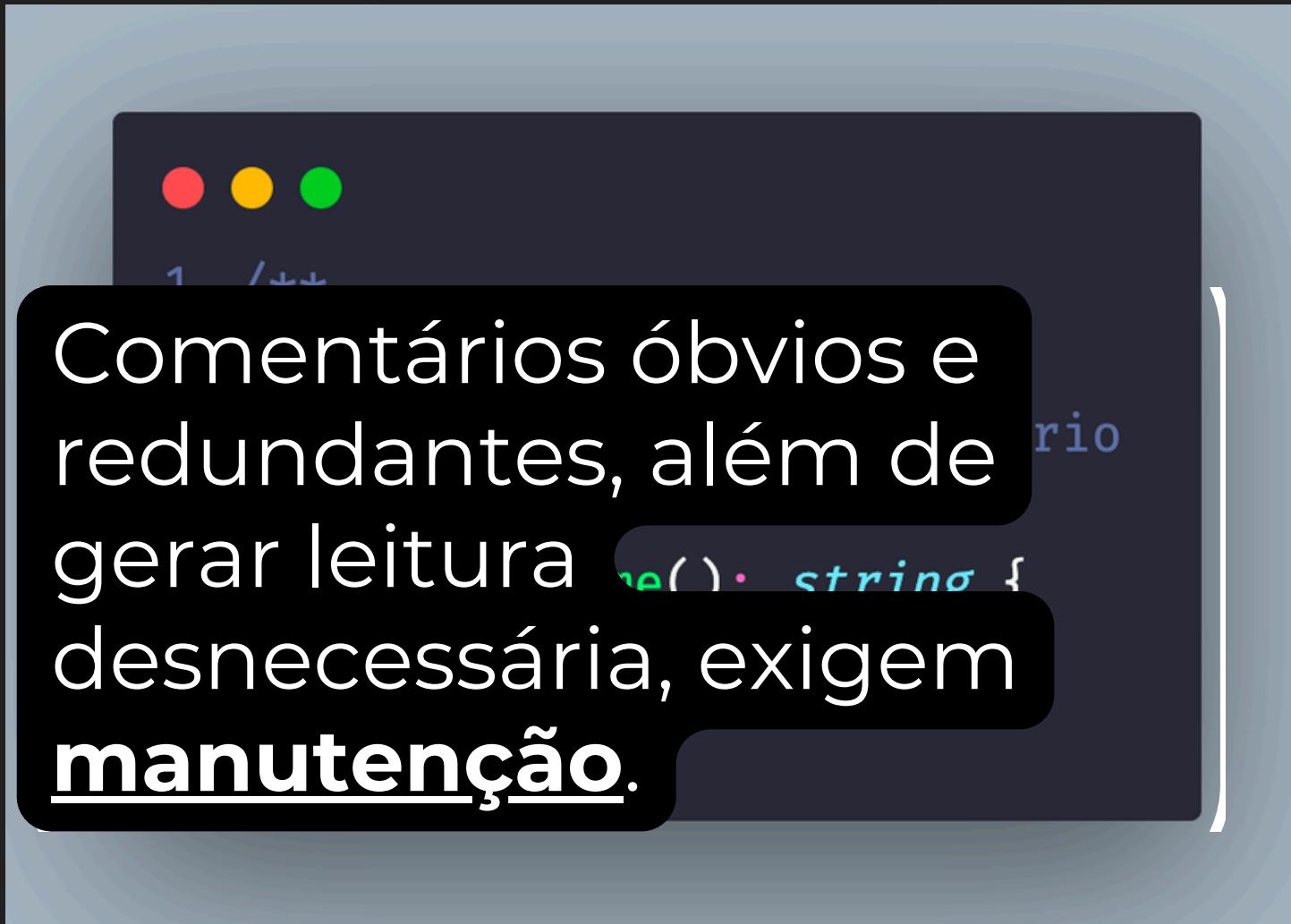
• Informações de autor

•

Comentários - Adicionar ou evitar

Comentar os métodos como:

 Adicionar



 Evitar

- Informações de autor
- Redundantes/ruidosos

Comentários - Adicionar ou evitar

Comentar rotinas como:



```
● ● ●  
1 //Verifica se o funcionário tem direito a todos benefícios  
2 if((employee.flags & HOURLY_FLAG) && (employee.age > 70)){  
3  
4 }
```



- Informações de autor
- Redundantes/ruidosos

Comentários - Adicionar ou evitar

Comentar rotinas como:



Não insira comentários em um código ruim, **reescreva-o**.

```
● ● ●  
1 //Verifica se o funcionário tem direito a todos benefícios  
2 if((employee.flags & HOURLY_FLAG) && (employee.age > 70)){  
3  
4 }
```



- Informações de autor
- Redundantes/ruidosos
- Comentar código confuso/ruim

```
● ● ●  
1 if((employee.isEligibleForFillBenefits())){  
2  
3 }
```



Comentários - Adicionar ou evitar

Comentar código com coisas a fazer
(TODO) como:

```
● ● ●  
1 async findOne(input: BaseInputWhere & FindOneOptions<User>): Promise<User> {  
2   const data = await this.userRepository.findOne({ ... input });  
3   if (!data) {  
4     //TODO: Trocar por exception ASAP  
5     return undefined;  
6   }  
7   return data;  
8 }
```

✓ Adicionar

✗ Evitar

- Informações de autor
- Redundantes/ruidosos
- Comentar código confuso/ruim

Comentários - Adicionar ou evitar

Comentar código com coisas a fazer
(TODO) como:

```
1 async findOne(input: BaseInputWhere & FindOneOptions<User>): Promise<User> {  
2   const data = await this.userRepository.findOne({ ... input });  
3   if (!data) {  
4     //TODO: Trocar por exception ASAP  
5     return undefined;  
6   }  
7   re1  
8 }
```

Nos ajudam, principalmente,
a anotar coisas que devem ser
implementadas no futuro,
que não podem ser feitas
agora.



Adicionar

- TODOs



Evitar

- Informações de autor
- Redundantes/ruidosos
- Comentar código confuso/ruim

Comentários - Adicionar ou evitar

Comentar código que não vamos usar
agora.

- ✓ Adicionar
 - TODOs

```
1 // async findOne(input: BaseInputWhere & FindOneOptions<User>): Promise<User> {  
2 //   const data = await this.userRepository.findOne({ ... input });  
3 //   if (!data) {  
4 //     return undefined;  
5 //   }  
6 //   return data;  
7 // }
```

- ✗ Evitar

- Informações de autor
- Redundantes/ruidosos
- Comentar código confuso/ruim

Comentários - Adicionar ou evitar

Comentar código que não vamos usar
agora.

- ✓ Adicionar
 - TODOs

```
1 // async findOne(input: BaseInputWhere & FindOneOptions<User>): Promise<User> {  
2 //   const data = await this.userRepository.findOne({ ... input });  
3 //   if (!data) {  
4 //     return undefined;  
5 //   }  
6 //   return data;  
7 // }
```

Uma das formas mais
abomináveis de comentários.
Revisores de código 'adoram'.

- ✗ Evitar

- Informações de autor
- Redundantes/ruidosos
- Comentar código confuso/ruim
- Comentários em código

Comentários

A reflexão é: Comentar código não é sinônimo de qualidade. O Código deve possuir comentário quando necessário, apenas.

Formatação

Formatação Vertical

Não daremos ênfase, mas observe a importância.

Listagem 5-1

BoldWidget.java

```
package fitnessse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
    public static final String REGEXP = "'''.+?''''";
    private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
        Pattern.MULTILINE + Pattern.DOTALL
    );

    public BoldWidget(ParentWidget parent, String text) throws Exception {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }

    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```

Listagem 5-2

BoldWidget.java

```
package fitnessse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
    public static final String REGEXP = "'''.+?''''";
    private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
        Pattern.MULTILINE + Pattern.DOTALL);
    public BoldWidget(ParentWidget parent, String text) throws Exception {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }
    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```

Recorte do livro: Código limpo: Habilidades práticas do Agile Software

Retirar essas linhas em branco, como na Listagem 5-2, gera um efeito notavelmente obscuro na legibilidade do código.

Recorte do livro: Código limpo: Habilidades práticas do Agile Software

Formatação Indentação

Não daremos ênfase, mas observe a importância.

```
public class FitNesseServer implements SocketServer { private FitNesseContext context; public FitNesseServer(FitNesseContext context) { this.context = context; } public void serve(Socket s) { serve(s, 10000); } public void serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new FitNesseExpediter(s, context); sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); } catch (Exception e) { e.printStackTrace(); } } }
```

```
public class FitNesseServer implements SocketServer { private FitNesseContext context; public FitNesseServer(FitNesseContext context) { this.context = context; } public void serve(Socket s) { serve(s, 10000); } public void serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new FitNesseExpediter(s, context); sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); } catch (Exception e) { e.printStackTrace(); } } }
```

Objetos e Estruturas de dados

Abstração

É dito que atributos devem ser **privados**, para que ninguém dependa diretamente deles.

Criar os getters/setters então, parece correto?



```
1 constructor(  
2   private _nome: string,  
3   private _forca: number,  
4   private _habilidadeMental: number,  
5   private _poderDeAtaque: number,  
6   private _esquiva: number,  
7   private _resistencia: number,  
8   private _vidaAtual: number,  
9   private _vidaMaxima: number  
10 ) {}  
11  
12 public get esquiva(): number {  
13   return this._esquiva;  
14 }  
15 public get vidaAtual(): number {  
16   return this._vidaAtual;  
17 }  
18 public get poderDeAtaque(): number {  
19   return this._poderDeAtaque;  
20 }
```

Caso Concreto e Abstração

Concreto, claramente sabemos o que possui um ponto, que tipo de ponto é, não existe abstração do funcionamento/características.

Abstrato, não sabemos o tipo de ponto, que tipo de coordenadas são, mas não deixa de representar um **Ponto**.



```
1 class Point{  
2     public x:number;  
3     public y:number;  
4 }
```



```
1 interface Point{  
2     getX():number;  
3     getY():number;  
4     setCartesian(x: number, y: number): void;  
5     getR(): number;  
6     getTheta(): number;  
7     setPolar(r: number, theta: number): void  
8 }
```

Isso é Orientação a Objetos?



```
● ● ●  
1 export class User extends BaseEntity {  
2  
3   name?: string;  
4   email?: string;  
5   password?: string;  
6   phone?: string;  
7   status?: Status;  
8   validationCode?: number;  
9   countRetrySms?: number;  
10  lastLogin?: Date;  
11  resetPasswordToken?: string;  
12  roles?: Role[];  
13 }
```

Isso é Orientação a Objetos?

É uma Estrutura de Dados!

Tanto que normalmente criamos um DTO (Data Transfer Object) acima dessas classes.

Costumam ser chamadas de

Active Records.

Está errado este formato?

```
1 export class User extends BaseEntity {  
2  
3   name?: string;  
4   email?: string;  
5   password?: string;  
6   phone?: string;  
7   status?: Status;  
8   validationCode?: number;  
9   countRetrySms?: number;  
10  lastLogin?: Date;  
11  resetPasswordToken?: string;  
12  roles?: Role[];  
13 }
```

Isso é Orientação a Objetos?

Está errado este formato?

Não! Errado seria **MISTURAR**.
Adicionar métodos, funções.
Uma Estrutura de dados deve ser
o mais **limpa** possível.

```
1 export class User extends BaseEntity {  
2  
3   name?: string;  
4   email?: string;  
5   password?: string;  
6   phone?: string;  
7   status?: Status;  
8   validationCode?: number;  
9   countRetrySms?: number;  
10  lastLogin?: Date;  
11  resetPasswordToken?: string;  
12  roles?: Role[];  
13 }
```



Se não
tem
teste,
não é
limpo

TESTE DE SOFTWARE

DESENVOLVIMENTO

X

TESTE DE SOFTWARE

DESENVOLVIMENTO

X

TESTE DE SOFTWARE



SE LIGA!

Teste de Software serve para provar que
o software não possui defeito.

SE LIGA! ERRO COMUM

~~Teste de Software serve para provar que
o software não possui defeito.~~

Serve para **ENCONTRAR DEFEITOS.**

POR QUE TESTAR?



LEI DE MURPHY

**"Qualquer coisa que possa ocorrer mal,
ocorrerá mal, no pior momento possível"**

TESTE DE SOFTWARE

Teste de software visa “destruir a aplicação”, encontrar todas as falhas possíveis.

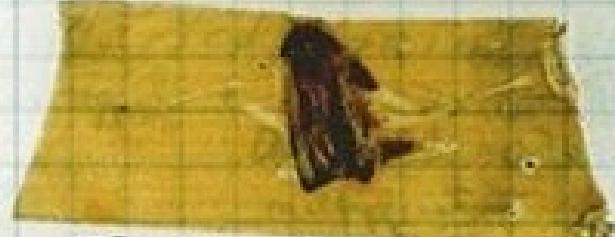


CURIOSIDADES

Origem do termo bug

Harvard ~1945

Inseto entre válvulas onde a retirada foi chamada de debug

17/06	9/9	
0800	Anton started	{ 1.2700 9.037 847 025
1000	stopped - anton ✓	9.037 846 ??5 connect
	13° C 033 MP-MC	1.982110900 2.130476415 (-3) 4.615925059 (-2)
	(033) PRO 2	2.130476415
	connect	2.130676415
	Relays 6-2 in 033 failed special sped test	Relay 2145
	in relay	2145 2147
1100	Started Cosine Tape (Sine check)	
1525	Started Multi Adder Test.	
1545		Relay #70 Panel F (moth) in relay.
1630	Anton just started.	First actual case of bug being found.
1700	closed down.	

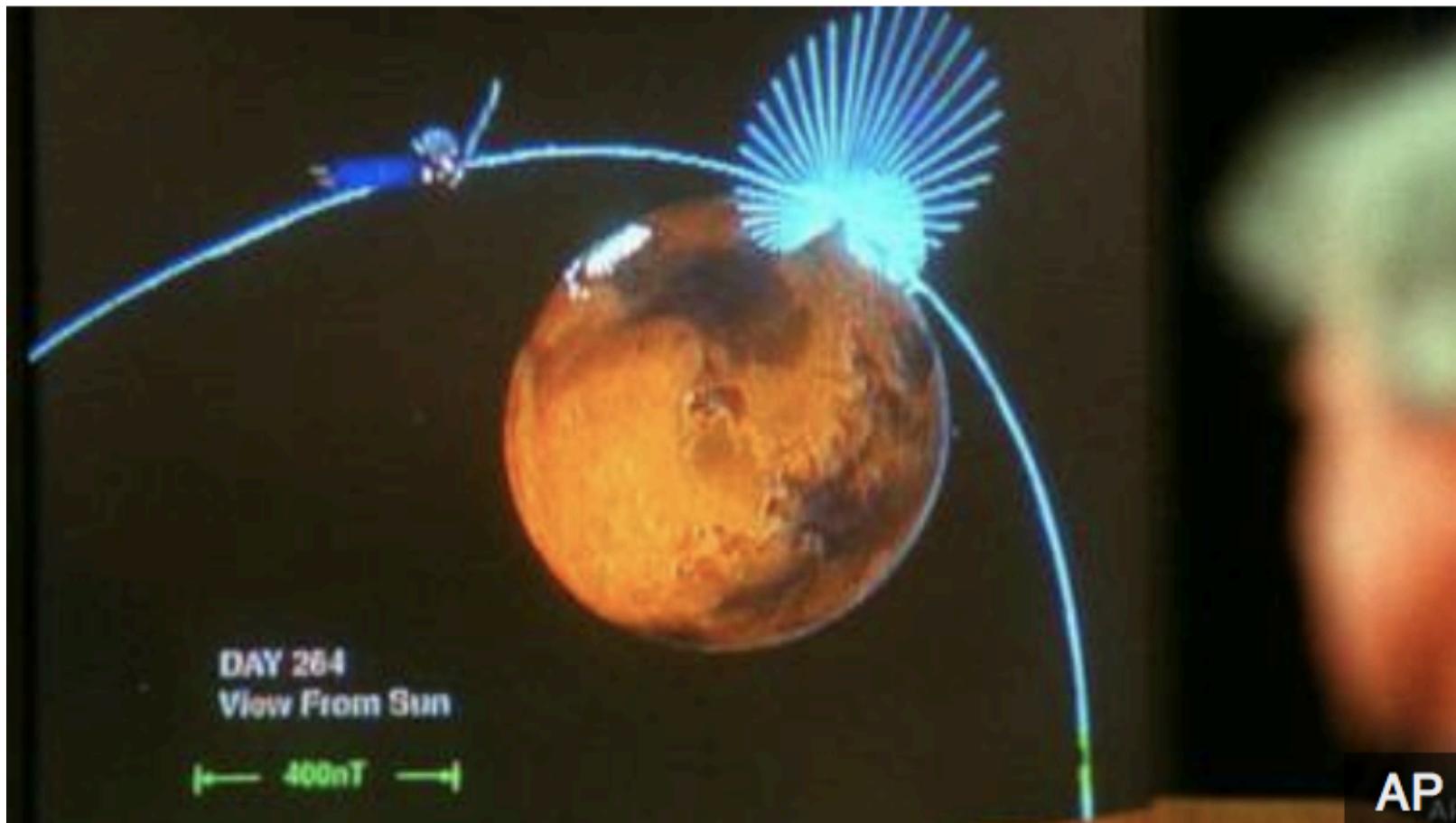
CURIOSIDADES

O Teste de Software custa entre 20% e 30% do investimento no desenvolvimento



CURIOSIDADES

1999
US\$125 MILHÕES



Satélite de US\$125 milhões desapareceu em 1999 por 'erro de conversão de unidades'

Outros erros da ciência/engenharia

https://www.bbc.com/portuguese/noticias/2014/05/140530_erros_ciencia_engenharia_rb

CURIOSIDADES

2014
2 MIL TRENS
US\$ 20,5 BI



Na França, novos trens são mais largos do que o tamanho da maioria das plataformas

Outros erros da ciência/engenharia

https://www.bbc.com/portuguese/noticias/2014/05/140530_erros_ciencia_engenharia_rb



CURIOSIDADES

1961
30 MORTOS



AFP

O navio Vesa afundou em sua viagem inaugural porque era mais espesso a bombordo

Outros erros da ciência/engenharia

https://www.bbc.com/portuguese/noticias/2014/05/140530_erros_ciencia_engenharia_rb



CURIOSIDADES - EXTINÇÃO

Hoje, existem poucos profissionais capacitados a trabalhar com Teste de Software, isso faz com que esse profissional seja bem valorizado.

Muito pouco tester raiz.



TESTER - PERFIL

Crítico

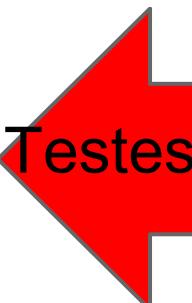
Visão voltada para a qualidade

Pessimista

Referência de qualidade na empresa

NÍVEIS DE TESTES / TESTES FUNCIONAIS

- Teste de Regressão
- Teste Beta
- Teste Alfa
- Teste de Aceitação
- Teste de Sistema
- Teste de Integração**
- Teste Unitário**



Testes realizados pelo programador (a nível de programação).

NÍVEIS DE TESTES / TESTES FUNCIONAIS

Teste de Regressão

Teste Beta

Teste Alfa

Teste de Aceitação

Teste de Sistema

Teste de Integração

Teste Unitário

Testes realizados pela equipe, após o sistema estar concluído, antes da entrega.

Testes realizados pelo programador (a nível de programação), durante o processo de desenvolvimento.

NÍVEIS DE TESTES / TESTES FUNCIONAIS

Teste de Regressão

Teste Beta

Teste Alfa

Teste de Aceitação

Teste de Sistema

Teste de Integração

Teste Unitário

Também a nível de sistema, mas feito pelo **usuário/cliente**.

Testes realizados pela equipe, após o sistema estar concluído, antes da entrega.

Testes realizados pelo programador (a nível de programação), durante o processo de desenvolvimento.

NÍVEIS DE TESTES / TESTES FUNCIONAIS

Teste de Regressão

Teste Beta

Teste Alfa

Teste de Aceitação

Teste de Sistema

Teste de Integração

Teste Unitário

Este realizado por grupos de pessoas conhecidas para avaliação de funcionalidades e pontos de melhorias. Sem planejamento.

Também é o nível de sistema, mas feito pelo cliente.

Testes realizados pela equipe, após o sistema estar concluído, antes da entrega.

Testes realizados pelo programador (a nível de programação), durante o processo de desenvolvimento.

NÍVEIS DE TESTES / TESTES FUNCIONAIS

Teste de Regressão

Teste Beta

Teste Alfa

Teste de Aceitação

Teste de Sistema

Teste de Integração

Teste Unitário

Este realizado por grupos de pessoas aleatórias para avaliação de funcionalidades e pontos de melhorias. Com report. Pré lançamento.

Teste realizado por grupos de pessoas conhecidas para avaliação de funcionalidades e pontos de melhorias. Sem planejamento.

Também a nível de sistema, mas feito pelo usuário.

Testes realizados pela equipe, após o sistema estar concluído, antes da entrega.

Testes realizados pelo programador (a nível de programação), durante o processo de desenvolvimento.

NÍVEIS DE TESTES

Teste de Regressão

Teste Beta

Teste Alfa

Teste de Aceitação

Teste de Sistema

Teste de Integração

Teste Unitário

Basicamente, consiste em re-executar os testes para verificar se o sistema não quebrou.

Teste realizado por grupos de pessoas aleatórias para avaliação de funcionalidades e pontos de melhorias. Com report. Pré lançamento.

Teste realizado por grupos de pessoas conhecidas para avaliação de funcionalidades e pontos de melhorias. Sem planejamento.

Também a nível de sistema, mas feito pelo usuário.

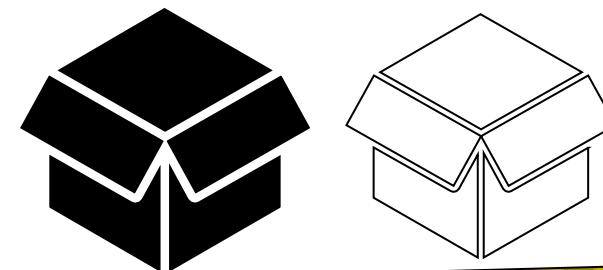
Testes realizados pela equipe, após o sistema estar concluído, antes da entrega.

Testes realizados pelo programador (a nível de programação), durante o processo de desenvolvimento.

TIPOS DE TESTES

- Teste de Regressão
- Teste Beta
- Teste Alfa
- Teste de Aceitação
- Teste de Sistema
- Teste de Integração
- Teste Unitário

ONDE CADA UM SE ENCAIXA?



Caixa Preta e Caixa
Branca

TIPOS DE TESTES





TESTE UNITÁRIO

**QUAIS AS
VANTAGENS DO
TESTE UNITÁRIO?**

VANTAGENS

Prevenção de bugs após entrega

Aumento da confiabilidade

Tranquilidade na modificação do código-fonte

Testa situações de sucesso e falha

Utilizado por diversas metodologias ágeis

Uma forma de validar requisitos

***EM QUE MOMENTO
REALIZAR OS
TESTES?***

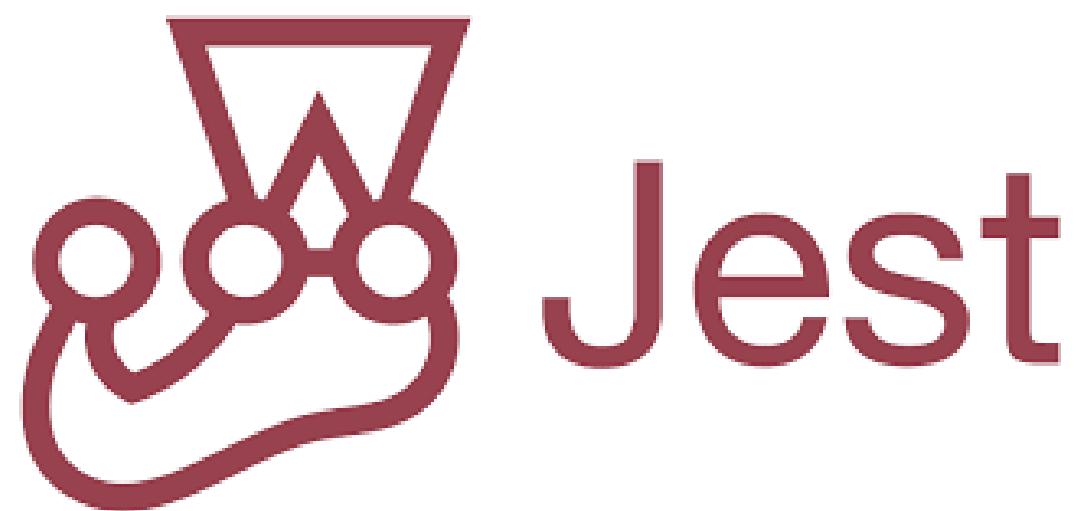
QUANDO

Antes de codificar a regra de negócios **e** diariamente para verificar se alguma atualização danificou os requisitos do sistema

**MAS E COMO
TESTAR?**



COMO TESTAR?



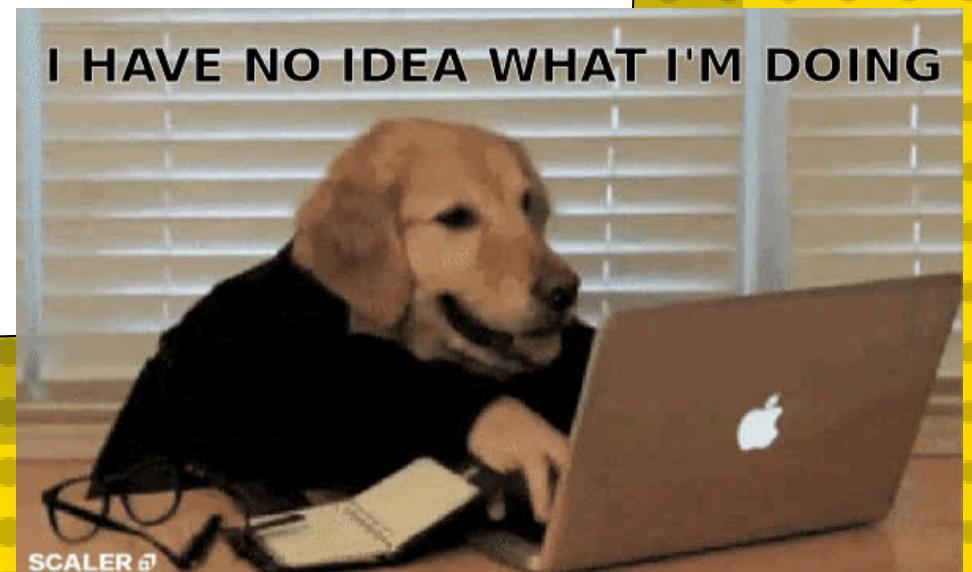
JUnit



**PLANEJAR,
EXECUTAR E
TESTAR**

PLANEJAR, EXECUTAR E TESTAR (EM 7 PASSOS)

1. Defina uma **lista com os testes** a serem implementados
2. Implemente uma **classe de teste** e desenvolva um **método** de teste para uma das tarefas
3. Rode o **teste** e certifique-se que o teste irá falhar
4. Implemente o código mais simples que rode o teste
5. **Refatore** o código para remover a duplicação de dados
6. Caso necessário, escreva mais testes ou aprimore o existente
7. Repita os passos de 2 a 6 até concluir toda a lista definida no item 1



Classes

Classes

Devem ser **pequenas**

- Pode ser uma classe **PEQUENA** de 100 linhas ou uma **GRANDE** de 15.



Classes

Devem ser **pequenas**

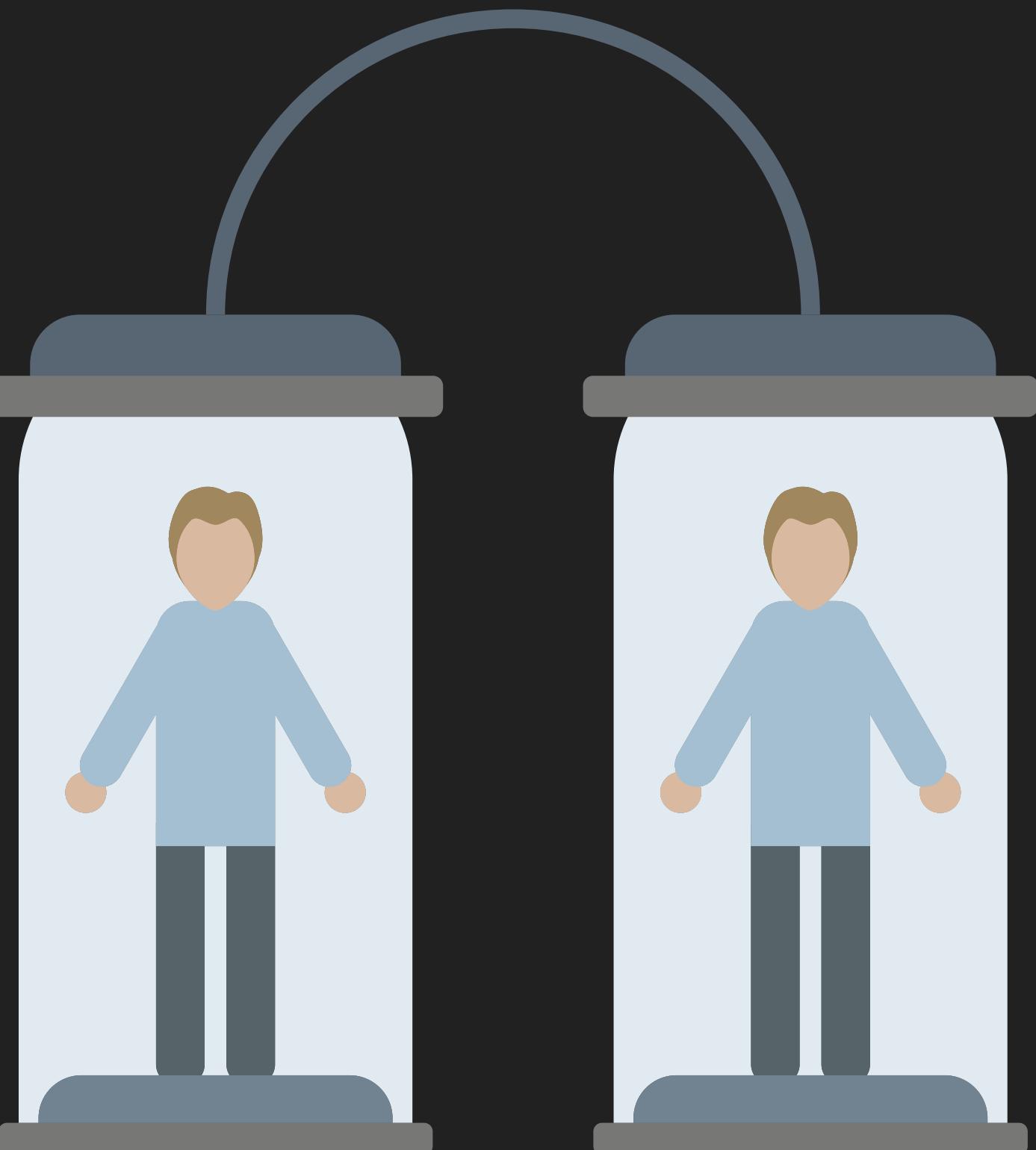
- Pode ser uma classe **PEQUENA** de 100 linhas ou uma **GRANDE** de 15.

A grandeza das classes é medida por **RESPONSABILIDADES**, e não por linhas de código.



Evitar Repetição (DRY - Don't Repeat Yourself)

Código duplicado é uma das principais fontes de problemas. Sempre que possível, reutilize código em vez de duplicá-lo.



Coesão - Classes precisam ser coesas

Classes são ditas coesas quando:

- Possui poucos atributos.
- Cada método da classe manipula pelo menos um atributo da classe.
- Quanto mais atributos um método manipular, mais coeso ele é para a classe.

Listagem 10-4

Stack.java - uma classe coesa.

```
public class Stack {  
    private int topOfStack = 0;  
    List<Integer> elements = new LinkedList<Integer>();  
  
    public int size() {  
        return topOfStack;  
    }  
  
    public void push(int element) {  
        topOfStack++;  
        elements.add(element);  
    }  
  
    public int pop() throws PoppedWhenEmpty {  
        if (topOfStack == 0)  
            throw new PoppedWhenEmpty();  
        int element = elements.get(--topOfStack);  
        elements.remove(topOfStack);  
        return element;  
    }  
}
```

Martin, Robert. Código limpo: Habilidades práticas de Agile Software.

A alta coesão indica que a classe possui seus atributos e métodos funcionando em torno de um propósito bem definido.

Coesão



```
1 class Produto {
2     private nome: string;
3     private preco: number;
4
5     constructor(nome: string, preco: number) {
6         this.nome = nome;
7         this.preco = preco;
8     }
9
10    obterNome(): string {
11        return this.nome;
12    }
13
14    obterPreco(): number {
15        return this.preco;
16    }
17 }
```

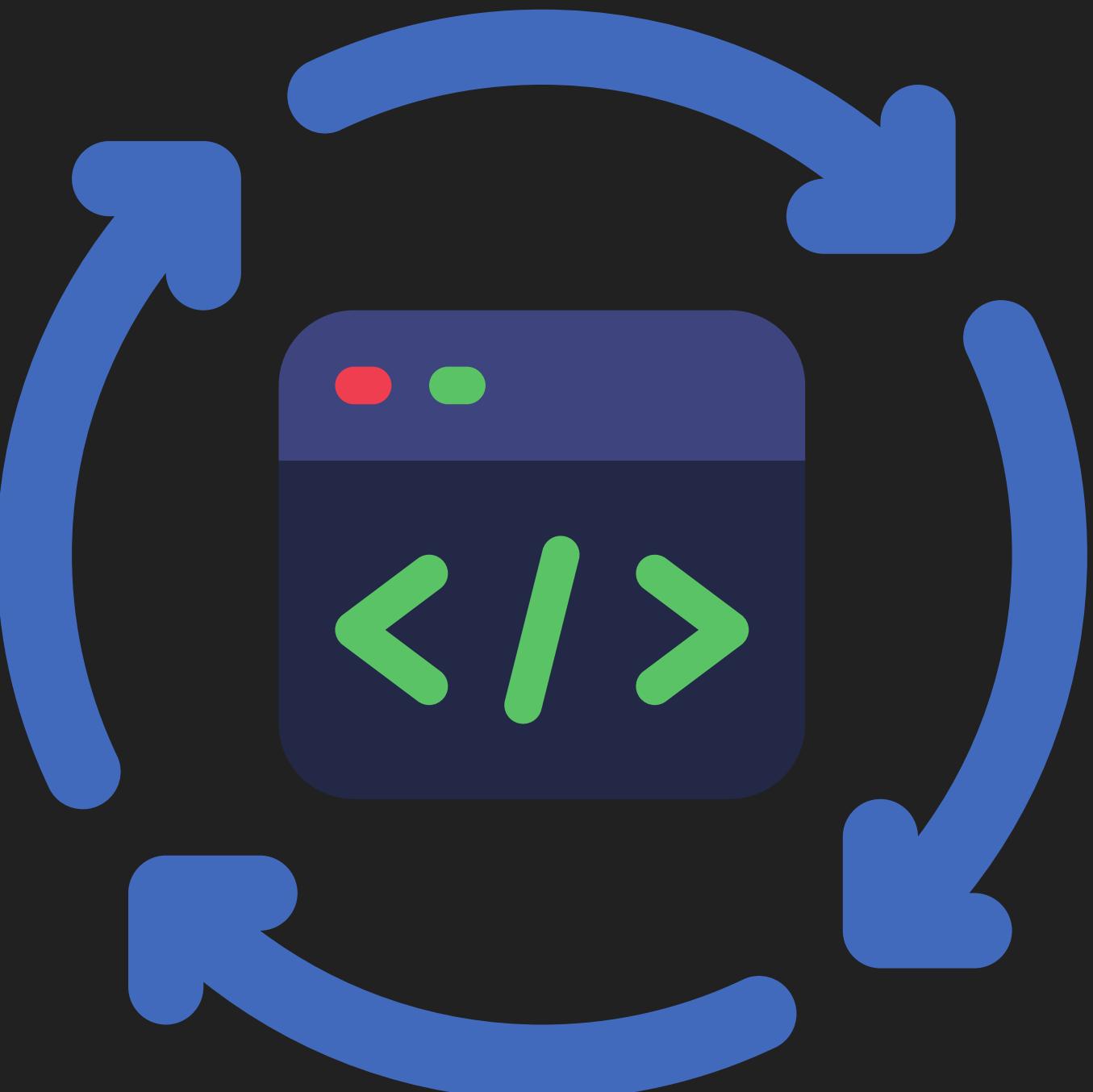
```
● ● ●
```

```
1  class Pedido {
2      private produtos: Produto[];
3      private total: number;
4
5      constructor() {
6          this.produtos = [];
7          this.total = 0;
8      }
9
10     adicionarProduto(produto: Produto): string {
11         this.produtos.push(produto);
12         this.total += produto.obterPreco();
13         return(`${produto.obterNome()} foi adicionado ao pedido. Preço: R${produto.obterPreco()}`);
14     }
15
16     removerProduto(produto: Produto): string {
17         const index = this.produtos.indexOf(produto);
18         if (index > -1) {
19             this.produtos.splice(index, 1);
20             this.total -= produto.obterPreco();
21             return(`${produto.obterNome()} foi removido do pedido.`);
22         } else {
23             return(`Produto não encontrado no pedido.`);
24         }
25     }
26
27     calcularTotal(): number {
28         return this.total;
29     }
30
31 }
```

Refatoração

Refatorar o código continuamente é uma prática essencial para manter o código limpo e **evitar a "dívida técnica"**.

A refatoração deve ser parte do fluxo de desenvolvimento, não um esforço esporádico.



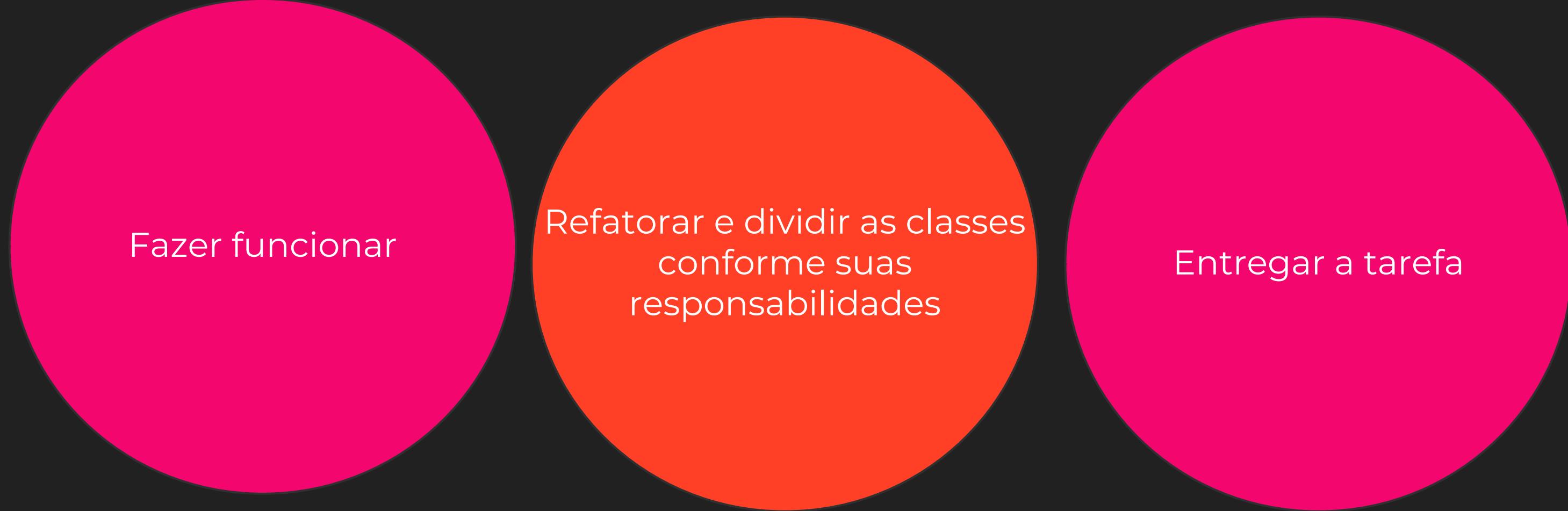
SOLID

5 principios para escrever código mais fácil de
manter, extender e modular

SRPolid - Princípio da Responsabilidade Única

As classes devem seguir o SRP

- **Uma** responsabilidade e **um** único motivo para mudar.



Fazer funcionar

Refatorar e dividir as classes
conforme suas
responsabilidades

Entregar a tarefa

Um dos principais conceitos da OO, dos mais simples, e dos mais deixados de lado.

SRPolid - Princípio da Responsabilidade Única

Criar muitas classes não pode acabar atrapalhando também o desenvolvimento do sistema?



SRPolid - Princípio da Responsabilidade Única

Criar muitas classes não pode acabar atrapalhando também o desenvolvimento do sistema?

Entende-se que, apesar do receio, os benefícios compensam o fato de ter muitas classes.

"É melhor ter uma caixa de ferramenta com muitas repartições pequenas com objetos bem classificados ou poucas gavetas com diversas coisas dentro?"



SRPolid - Princípio da Responsabilidade Única

Aqui, a classe Order é responsável apenas por gerenciar os itens do pedido.

A classe Logger é responsável por registrar os logs. Isso evita que a classe Order tenha mais de uma responsabilidade.

```
class Logger {
  log(message: string): void {
    console.log(message);
  }
}

class Order {
  private items: string[];
  private logger: Logger;

  constructor(items: string[], logger: Logger) {
    this.items = items;
    this.logger = logger;
  }

  addItem(item: string): void {
    this.items.push(item);
    this.logger.log(`Item ${item} added to the order.`);
  }
}
```

SOCPlid - Princípio Aberto/Fechado

Entidades de software (classes, módulos, funções) devem estar **abertas** para extensão, mas **fechadas** para modificação.

Isso significa que devemos ser capazes de adicionar **novas funcionalidades sem alterar** o código existente.

SOCPLid - Princípio Aberto/Fechado

A classe Order **não precisa ser modificada** para aplicar novos tipos de desconto.

Podemos criar novas implementações da interface Discount e injetar a nova lógica sem alterar a classe Order.

```
interface Discount {
    apply(price: number): number;
}

class NoDiscount implements Discount {
    apply(price: number): number {
        return price;
    }
}

class TenPercentDiscount implements Discount {
    apply(price: number): number {
        return price * 0.9;
    }
}

class Order {
    private price: number;
    private discount: Discount;

    constructor(price: number, discount: Discount) {
        this.price = price;
        this.discount = discount;
    }

    calculateTotal(): number {
        return this.discount.apply(this.price);
    }
}
```

SOLSPid - Princípio da Substituição de Liskov

Se **S** é um subtipo de **T**, então objetos do tipo **T** devem **poder ser substituídos** por objetos do tipo **S** sem quebrar a aplicação.

Isso significa que uma subclasse deve comportar-se de maneira consistente com a classe base.

SOLSPid - Princípio da Substituição de Liskov

O **Penguin**, que herda de **Bird**, não se comporta como os outros pássaros, quebrando o código.

Para resolver isso, é melhor evitar esse tipo de herança se o comportamento não puder ser mantido.

```
class Bird {
  fly(): void {
    console.log("Flying...");
  }
}

class Penguin extends Bird {
  fly(): void {
    throw new Error("Penguins can't fly!");
  }
}

function letBirdFly(bird: Bird): void {
  bird.fly();
}

const penguin = new Penguin();
letBirdFly(penguin); // Irá quebrar, violando o princípio LSP.
```

SOLISPd - Princípio da Segregação de Interfaces

Os clientes **não devem ser forçados** a depender de interfaces que não utilizam.

Em vez de interfaces grandes e genéricas, devemos criar interfaces menores e mais específicas para cada tipo de cliente.

SOLISPd - Princípio da Segregação de Interfaces

Nesse exemplo, temos duas interfaces separadas, **Worker** e **Eater**.

O robô não precisa implementar o método `eat()`, respeitando o princípio ISP.

```
interface Worker {  
    work(): void;  
}  
  
interface Eater {  
    eat(): void;  
}  
  
class Robot implements Worker {  
    work(): void {  
        console.log("Robot is working.");  
    }  
}  
  
class Human implements Worker, Eater {  
    work(): void {  
        console.log("Human is working.");  
    }  
  
    eat(): void {  
        console.log("Human is eating.");  
    }  
}
```

SOLIDIP - Princípio da inversão de dependências

Dependa de **abstrações**, não de implementações concretas.

Módulos de alto nível não devem depender de módulos de baixo nível diretamente. Ambos devem depender de abstrações (interfaces).

SOLIDIP - Princípio da inversão de dependências

A classe OrderService depende da interface Database, e não de uma implementação específica como MySQLDatabase.

Isso facilita a troca de bancos de dados sem alterar a lógica da aplicação.

```
interface Database {
  save(data: string): void;
}

class MySQLDatabase implements Database {
  save(data: string): void {
    console.log(`Saving data to MySQL: ${data}`);
  }
}

class OrderService {
  private database: Database;

  constructor(database: Database) {
    this.database = database;
  }

  saveOrder(data: string): void {
    this.database.save(data);
  }
}

const mySQLDatabase = new MySQLDatabase();
const orderService = new OrderService(mySQLDatabase);
orderService.saveOrder("Order data");
```



Atividade

Link para a atividade:

- [https://docs.google.com/document/d/1HRR0jfTBJRb0hLibXt1Ozjba45cSqb0eFoORbUXTtlS/edit?
usp=sharing](https://docs.google.com/document/d/1HRR0jfTBJRb0hLibXt1Ozjba45cSqb0eFoORbUXTtlS/edit?usp=sharing)
-

